

Beyond Simple Transactions and Atomic Blocks

Victor Luchangco
Sun Microsystems Laboratories
victor.luchangco@sun.com

Concurrent programming is notoriously difficult because actions by concurrent threads may be interleaved and result in unanticipated interactions, particularly on a shared-memory system. One mechanism proposed to address this difficulty is *transactional memory* (TM): programmers can group several operations into a *transaction*, which a TM implementation guarantees will appear to execute atomically,¹ or not at all (i.e., the transaction is *aborted*). Thus, transactions reduce the possible interleavings and interactions that the programmer must consider.

With transactional memory, it is easy to implement *atomic blocks*: one can simply execute the atomic block within a transaction, retrying in case the transaction aborts. Atomic blocks are typically implemented using locks to protect accessed data, in which the programmer must choose how to associate locks with data—trading simplicity against permissible concurrency—and manage the well known challenges of lock-based programming (e.g., the possibility of deadlock). Using transactions rather than locks, raises the level of abstraction, and shifts the responsibility of ensuring apparently atomic execution from the programmer to the system. Thus, programmers do not need to acquire the locks protecting the accessed data (and in such a way that avoids deadlock). Indeed, a TM implementation might not use locks to guarantee atomicity, but rather execute transactions optimistically, and abort those that will not appear atomic.

However, transactional memory is not a panacea for concurrent programming: Many concurrent programs will require different kinds of synchronization, perhaps to avoid the overhead of implementing transactions or because another kind of synchronization is more “natural” in certain cases. For example, one thread may have to wait for another to compute a value (i.e., a producer-consumer pair). Or a thread may want to do some I/O (I/O is synchronization with the “user”). In addition, transactions may interact with other programming constructs even for a single thread, such as exceptions. For TM to be successful, its behavior with respect to these other kinds of synchronization and programming constructs must be well-defined, and preferably easy to understand. Some people suggest that transactional semantics should be defined in terms of locking semantics [13], and that TM should be considered a technique for implementing the semantics of a single global lock, but allowing greater concurrency than would be implied by actually having a single lock [3]. Although this may yield many of the performance benefits of TM, it gives up the conceptual benefits of TM, which I believe are ultimately more important.

Key among these benefits is modularity: A transaction hides within itself the details of how it is implemented. The lack of this ability to hide implementation details is one of the chief reasons that lock-based programming is difficult: to avoid deadlock, a programmer must typically expose what locks may be acquired in a critical section, making the locks are part of the interface, not just the implementation. I hope that transactions can be to concurrent programming what functions are to sequential programming: the basic building blocks from which programs are constructed. This

¹As discussed below, transactional memory implementations differ on exactly what is encompassed by this guarantee.

motivation has far-reaching consequences. For example, if transactions are the basic building block in a system where new threads can be generated dynamically, then a transaction must be able to support multiple threads within it: that is, parallelism must be allowed within transactions. The Fortress programming language allows this [2], but few TM implementations do. Similarly, I/O is typically forbidden within transactions, but if transactions are the basic building blocks, we must find some way of making sense of I/O within transactions. I believe the formal methods community can make a real contribution in guiding how transactions ought to be defined formally so that we can reason rigorously about them.

There already exists some work that explores how the transactional interface can be extended beyond simple flat transactions or atomic blocks, or even “closed nested” transactions. For example, the original dynamic software transactional memory (DSTM) [7] provided an *early release* mechanism. However, this mechanism was described in operational terms, rather than giving an abstract specification of its guarantees, and so it would be hard to reason about this without modeling the DSTM implementation. *Open nested transactions* allow some operations within a transaction to become visible before the transaction completes. Initially, it too was described operationally, but various methods have been developed to reason about structured uses of open nesting [1, 14]. *Transactional boosting* [6] is a methodology for “wrapping” a class of linearizable implementations of objects that don’t use transactions (and therefore don’t pay the overhead for such use) so that they can be used within transactions. *Abstract nested transactions* [5] provide a way to designate parts of a transaction as safe to re-execute at the end of the transaction in case the data accessed by the parts so designated are changed by some other thread before the transaction completes, and if the result is the same as before, the transaction need not abort. All these mechanisms allow more efficient implementation within the transaction, but leave the basic transactional abstraction intact at the boundary of the outermost transaction (assuming that they are used correctly).

To allow I/O, for example, we need to extend the transactional interface even at the boundary of the outermost transaction. One approach is to allow *irrevocable transactions* [16], which cannot be aborted. However, to ensure that such transactions can complete successfully, at most one irrevocable transaction can be executing at any time, which greatly restricts its utility. I think we need some more flexible mechanism, one that perhaps exposes the input and output streams (so the streams become part of the interface), which may change while the transaction executes (i.e., the transaction’s effects on these streams might not be atomic). Effects on these streams might be buffered, so that an aborted transaction leaves them unchanged (at least abstractly). To prevent input that depends on output, perhaps the input streams only yield input that was available at the logical beginning of the transaction. Clojure [8] provides *agents* that have some of these properties (but see below for more discussion on Clojure).

We might also want to support condition variables within transactions; this requires breaking the simple transactional abstraction, which does not support any notion of “waiting”. Dudnik and Swift [4] propose a scheme similar to early work on monitors, in which waiting causes the transaction to complete and a new transaction to be started when the condition becomes true. Does their scheme support the kind of modularity we desire?

Transaction synchronizers [11] are another mechanism that expand the transactional interface, this time by allowing multiple transactions to interact through special synchronizer objects. Although interacting transactions do not, of course, appear atomic to each other, they can interact only through the synchronizers, and furthermore, transactions that interact appear to be all a single transaction to other threads, so the extent about which the programmer must reason about is limited. However, with synchronizers, a programmer must determine which transactions interact with each other (and therefore must either all complete successfully or else all abort).

An explicit abort facility is also often proposed for transactional memory; that is, a facility by

which the programmer can explicitly abort a transaction based on what it reads. Such a facility enables Dijkstra-style guarded commands to be encoded straightforwardly as a transaction even when computing the guard is complicated and involves writing memory: the transaction can be simply aborted if the guard turns out to be false. Doing so without an abort facility would require the programmer to compute the guard separately in a scratch space and then write the changes back to the appropriate locations if the guard turns out to be true, which would be longer and more wasteful in time and space.

Privatization is another area that has received a lot of attention recently [9, 12, 15], with a lot of work on supporting *implicit* privatization, just as if transactions were implemented with locks. Despite progress in this regard, it is still substantially expensive to support implicit privatization. However, we can avoid this cost if we extend the transactional interface to require transactions to declare when they might privatize a chunk of memory.

We could also weaken transactional guarantees by allowing concurrent transactions to appear to different threads to be done in different orders. In my thesis [10], I gave a couple of examples of such weakly consistent models. The relaxed requirements could allow implementations much more flexibility. Whether the improvement in efficiency enabled by this flexibility justifies the added complexity to the TM user needs to be studied. Because weak consistency models are tricky to reason about, a formal model would be particularly helpful.

Another way we might extend the transactional interface is to add priorities. Note that this is different from a contention management policy, which probably makes most sense to leave to the TM implementation. Rather, the programmer might assign priorities to transactions, which would be used by the contention manager to schedule transactions and to decide which ones to abort when conflict arises.

These last several examples also illustrate why I think lock-based semantics, even just "single global lock atomicity" (SGLA) semantics, is a poor choice for TM. Under SGLA, transactions cannot be concurrent, and so cannot interact using synchronizers; there is no notion of aborting; implicit privatization is automatically safe; transactions must be strongly consistent; and transactional priorities don't make sense because once a transaction has begun, it does not relinquish its "lock" until it is done. Undoubtedly, it is easiest to convert lock-based programs to use TM if the semantics of TM is as close to locks as possible, but we already know that lock-based programming is hard, and does not provide the modularity that transactions promise. I think it is better to formulate a new model for transactions that captures the intuition of "atomicity" or "isolation", and then use that as a basis for extending the transactional interface to work with various kinds of synchronization and admit efficient implementations.

Finally, this paper has only considered TM in the context of an imperative programming style. But to really exploit large-scale multiprocessors, it will probably be necessary to adopt a more functional style of programming, with less shared mutable state. It is important to consider how TM would fit in to such a model, and how that changes certain trade-offs.

In conclusion, I think we need to start thinking about how TM will fit in with other kinds of synchronization, and other models of parallel computation. I've posed several examples of ways we might extend the interface and raised some issues that arise in doing so, but I make no claims about what is the *right* way to do so. I think this must be guided by experience, and also by what we can say precisely and reason about rigorously.

References

- [1] Kunal Agrawal, I-Ting Angelina Lee, and Jim Sukha. Safe open-nested transactions through ownership. In *PPoPP*, 2009.
- [2] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan Maessen, Sukyoung Ryu, Guy Steele, and Sam Tobin-Hochstadt. *The Fortress Language Specification, Version 1.0*. Sun Microsystems, Inc., 2008.
- [3] Hans-J. Boehm. Transactional memory should be an implementation technique, not a programming interface. In *HotPar*, 2009.
- [4] Polina Dudnik and Michael M. Swift. Condition variables and transactional memory: Problem or opportunity? In *Transact*, 2009.
- [5] Tim Harris and Srdjan Stipic. Abstract nested transactions. In *Transact*, 2007.
- [6] Maurice Herlihy and Eric Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *PPoPP*, 2008.
- [7] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC*, 2003.
- [8] Rich Hickey. Clojure. <http://clojure.org>.
- [9] Yossi Lev, Victor Luchangco, Virendra Marathe, Mark Moir, Dan Nussbaum, and Marek Olszewski. Anatomy of a scalable software transactional memory. In *Transact*, 2009.
- [10] Victor Luchangco. *Memory Consistency Models for High Performance Distributed Computing*. ScD thesis, MIT, 2001.
- [11] Victor Luchangco and Virendra Marathe. Transaction synchronizers. In *SCOOOL*, 2005.
- [12] Virendra Marathe, Michael Spear, and Michael Scott. Scalable techniques for transparent privatization in software transactional memory. In *ICPP*, 2008.
- [13] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Bratin Saha, and Adam Welc. Practical weak-atomicity semantics for Java stm. In *SPAA*, 2008.
- [14] Yang Ni, Vijay Menon, Ali-Reza Adl-Tabatabai, Antony Hosking, Richard Hudson, Eliot Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *PPoPP*, 2007.
- [15] Michael Spear, Virendra Marathe, Luke Dalessandro, and Michael Scott. Privatization techniques for software transactional memory. Technical Report 915, U. of Rochester, 2007.
- [16] Adam Welc, Bratin Saha, and Ali-Reza Adl-Tabatabai. Irrevocable transactions and their applications. In *SPAA*, 2008.